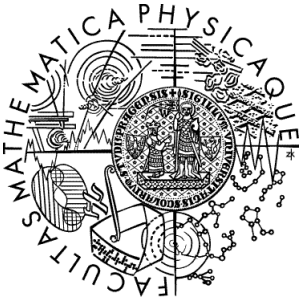
Charles University in Prague
Faculty of Mathematics and Physics

# BACHELOR THESIS




Michal Antolík

## Systém pro vytváření a kompilaci programů v grafickém daty řízeném paralelním programovacím jazyce

### The System for Creation and Compilation of Programs in a Graphical Dataflow Parallel Programming Language

Department of Software Engineering

Supervisor: RNDr. Petr Hnětynka, Ph.D.

Study Program: Computer Science

2009

I would like to thank my former supervisor, Mgr. Jiří Vyskočil, Ph.D., for suggesting me the idea and giving me numerous advice. I would also like to thank my latter supervisor, RNDr. Petr Hnětynka, Ph.D, for helping me with finishing the work on the project.

Furthermore I would like to thank my parents for supporting me in my work.

I hereby declare that I wrote the thesis myself using only the referenced sources. I agree with lending of the thesis.

Prague, May 19, 2009 Michal Antolík

# Contents

# List of Figures

Název práce: Systém pro vytváření a kompilaci programů v grafickém daty řízeném paralelním programovacím jazyce
Autor: Michal Antolík
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí bakalářské práce: RNDr. Petr Hnětynka, Ph.D.
e-mail vedoucího: hnetynka@dsrg.mff.cuni.cz

Abstrakt: Cieľom tejto práce je navrhnúť schému pre grafickú tvorbu dátmi riadených paralelných procesov naprogramovaných v jazyku JAVA, implementovať editor pre vizuálne vytváranie daných schém a súčasne vykonávacie jadro, ktoré zabezpečí ich spustenie podľa vopred definovaných pravidiel.
Súčasťou textu práce je vysvetlenie data-flow paradigmy a jeho využitia pri paralelných procesoch, analýza spojenia data-flow konceptu a jazyka JAVA ukázana na tvorbe cyklov, t.j. vetvenie a spájanie rôznych dátových prúdov s prihliadnutím na vznik nedeterminizmu, porozumenia a prehľadnosti výsledných inštancií schémy. Rovnako súčasťou textu je aj popis navrhovaného riešenia, implementácie jednotlivých častí , uživateľská dokumentácia s jednoduchými príkladmi a porovnanie existujúcich data-flow nástrojov s navrhovaným riešením.

Klíčová slova: data-flow, paralelné procesy, vizuálne programovanie, JAVA

Title: The System for Creation and Compilation of Programs in a Graphical Data-flow Parallel Programming Language
Author: Michal Antolík
Department: Department of Software Engineering
Supervisor: RNDr. Petr Hnětynka, Ph.D.
Supervisor's e-mail address: hnetynka@dsrg.mff.cuni.cz

Abstract: The goal of the thesis is to design a scheme for the graphical composition of data-flow parallel processes developed in JAVA programming language, implement an editor for their visual creation and also the engine core for their execution according to the predefined rules.
The thesis also presents a description of the data-flow paradigm and its use with parallel processes, an analyze of data-flow concept conjucted with JAVA language in the way of loop creation, i.e. branching and merging of different data flows, and its impact on an indeterminism, lucidity and understanding of scheme instances. The thesis describes the proposed solution and its implementation, the user documentation with simple examples and the overview of existing data-flow tools.

Keywords: data-flow, parallel processes, visual programming, JAVA

# Chapter 1

# Introduction

The development of processors indicates the fact that increase of a clock rate is slowing down due to heat dissipation issues, therefore the overall power of processors will be increased by introducing of new larger number of cores. Thus it means, that speed of a program execution can be improved only in case of sufficient scalability when computations are divided into several threads. However, concurrent programming with many threads can produce poor transparency in a program run-time and eventually difficult debugging when some problems occured. Another disadvantages are requirement of good knowledge of mastering thread communication and using synchronization tools as well as to have sufficient overview of tasks scheduling to avoid deadlocks and non determinism.

Back in the history, with exploitation of massive parallelism since the 1960s, there had been a hardware specialized in concurrent task execution which idea is derived from data-flow paradigm. However, programs already written for von Neumann processor architecture were not suitable to be compiled on data-flow hardware, therefore consequent development of concurrent task execution was rather intended for computers with von Neumann architecture.

In the last years, several imperative languages with native support of parallelism have been established, among them belongs also popular Java programming language. Primary motivation of this project is to analyze possibilities of conjuction data-flow concept with Java language. Design of a program will be divided in two parts to increase an overall readibility[1] of programs created according to a data-flow concept. First one will represent a visual design of data flows among tasks.The second part represents an implementation of all tasks functionality in Java language without using threads and synchronization tools needed for concurrent programming. There will be also implemented execution engine to run these data-flow programs. This engine will be a part of this project, it will be implemented in the Java language and finally and it will keep programmers from understanding all matters of parallel programming.

## 1.1 Project structure

The section 2.1 describes what a data-flow paradigm means, its properties and the reason why it is advantageos in parallel processes systems. The properties of

[1] According to [10], time spend in development of project in visual data-flow language was significantly faster than in C, mainly due to visual syntax.

this concept are compared to a concept of Java language, and consequently some inconsistences and restrictions are concluded. In next section 2.2, the several approaches of scheduling data-flow instructions are inspected. The analyze is specialized in a loop visual representation and shifting data among particular iterations. Several proposals and its properties are compared. Also, the most suitable proposal isselected and it is described in detail in subsequent chapter 3.

Next chapter 4 describes framework used for definition of proposal from previous chapter, an implementation process of data-flow scheme editor, but also the reason, why particular framework is used for a development. As well, important aspects in an implementation of execution engine are reported in this chapter. Chapter 5 is intended for description a way how to create and run data-flow programs.

Short description of related projects and its comparision to suggested solution is written in chapter 6, the conclusion and the evaluation of the the project and future work are described in chapter 7.

# Chapter 2

# Analyse of conjuction data-flow paradigm with Java language

## 2.1 Introduction to data-flow paradigm

The data-flow paradigm comes from the conceptual notion that a program is a directed graph and that data flows between the *instructions*, along the *arcs*. Since the goal is to apply a data-flow concept into Java language on multithreaded von Neumann architecture, the *instruction* is meant to be a block of sequentially written commands - the function from imperative languages with several inputs and outputs. The output from one data-flow instruction and input from another is connected by an *arc*[1] which symbolize the flow of data and represent the queue (*FIFO*). The best list of features that constitute a data-flow language was put forward by Ackerman in 1982 and it includes the following rules:

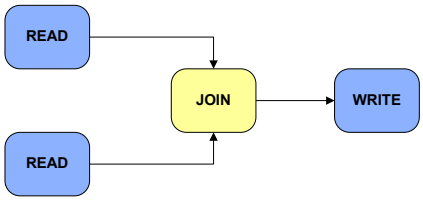



Figure 2.1: Simple dataflow model showing one iteration where tokens from two read stream are merged and put to the output stream

1. freedom from side effects
2. locality of effect
3. data dependencies equivalent to scheduling
4. single assignment of variables
5. lack of history sensitivity in procedures

Keeping these points in a language design will guarantee a desirable scalability and consequently the scheduling of data-flow instructions in the run-time engine. In a pure data-flow language these points means:

- The scheduling is determined from data dependencies, therefore it is important that the value of variables do not change between their definition and

[1] also will be called as *wire* or simply *conection*

their use. This results in fact that the variables can be regarded as values, rather than references to an objects (pointer to value or object). The implication of the single assignment rule is that each value can be represented as an arc in the resultant data-flow graph, going from the instruction that assigns the value to the instruction that uses that value.

- An important consequence of the single assignment rule is that the order of statements in a data-flow language is not important. Provided there are no circular references, the definitions of each value or variable can be placed in any order in the program. The order of statements become important only when a loop is being defined.

- Freedom from side effects is also essential if data dependencies are to determine scheduling. Commonly, it is done by disallowing global variables and introducing scope rules. The lack of history in procedures is important, because of possibility to schedule the same data-flow instructions from different iterations at various execution units. Therefore, operations in shared memory are thread safe.

However, applying these rules to Java programming language can conflict with the following properties:

- static variables (operation to an object in static memory should not be a thread safe)

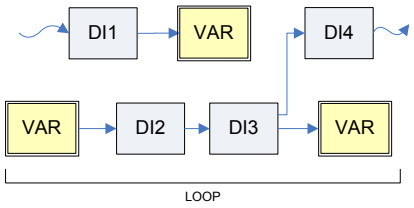
- singleton design pattern (to use only one instance for certain class)

- variable is not a value but a reference to an object (except primitive types)

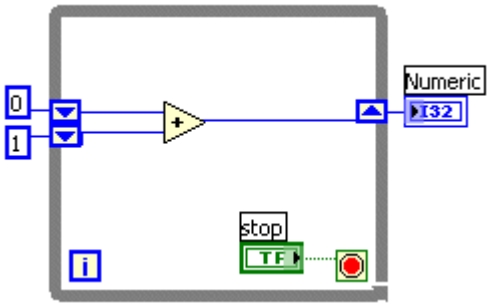
These properties are considered as main inconsistences with data-flow paradigm and in certain way they can break all five properties listed as features above. Some effects can be partialy checked, e.g. by checking equality of objects references which flow from the same data-flow instruction in the same time and consequently duplication can be done here, although, still a reference to an object can be saved in a singleton class instance. And since a Java code written sequentially inside blocks can be hardly inspected that it is „correctly" developed and it can not leave footprints after processing (that a block of code is interpreted as one independent single data-flow instruction), therefore it will be assumed that the code represents instruction is written properly.

## 2.2 The analyse of possible approaches in scheduling of data-flow instructions

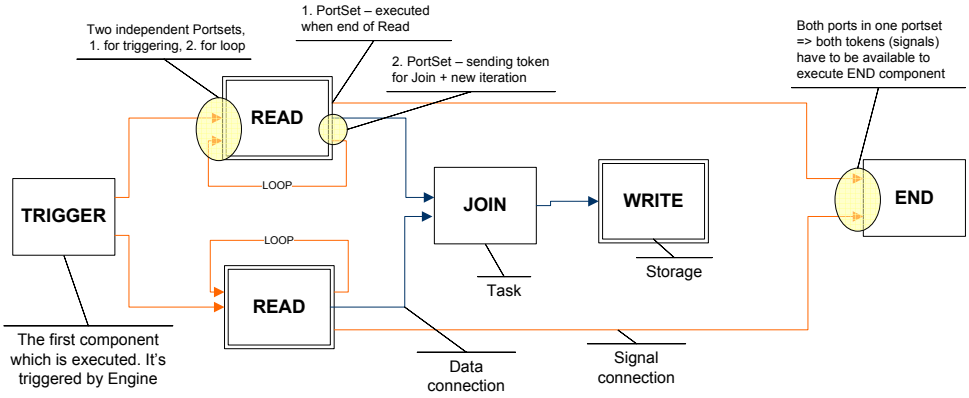
In this section several approaches of scheduling instructions dependant on data availability will be inspected, with an orientation to solve a loop representation in visual data-flow languages. Another open question there is how to shift data among loop iterations and how to initialize them before first iteration. The analyse will be based on several proposals and comparision of their properties.




(a) Proposition of a loop with global stack like variable




(b) Labview loop solution with shift registers




(c) Solution containing only nodes and arcs

Figure 2.2: Loop and passing values proposals

### 2.2.1 Shifting data among loop iterations

When a loop is designed, it is necessary to solve a way, how to switch from the initial to the produced data, and how to pass data from the previous to the next iteration. For a loop in an imperative language it is quite common to have some variables outside the loop, which are used to store partial results and consequently these data are used in a new iteration. However, in a pure data-flow language there is not a global memory and all operations have only local effect (afer operation no footprints lefts). The another condition in a loop design is that the loop should be displayed visually in an understanding way.

As a solution, these possibilities have arisen:

1. to have one value stack component - global variable - representing place in a memory which is controlled by synchronization locks due to access from various threads. It has two states denoting the existence of value. A value to this variable can be assigned at various places, but it can be read only at one place. The reason is that as soon as value is assigned, the particular value is propagated to the connected arc. See figure 2.2a. First, from DI1 (dataflow instruction 1) a value is assigned to VAR. This value is immediately delivered to DI2 (from the only one output VAR) and consequently to DI3 where a decision is make, if a new iteration should be performed (value is sent to VAR again) or the loop should be finished (value is sent to DI4).

A node is ready to be executed as soon as each input receive a token.

2. in the Labview[8] application, an inner part of a loop is enfold to a block with special elements (named shift registers) on both sides. These registers are used to shift values from the end of iteration to the new one. See figure 2.2b for the while loop example. Before first iteration, some defaults values are assigned to the shift registers and at the end of each iteration new values are passed throu registers to the next one (data are flowing from left to right). Specially at figure 2.2b, there are shown stacked shift registers which have an ability to remember values from the last two iterations, with the most recent iteration value stored in the top shift register. The loop finish when a true boolean token is delivered to the right bottom element (red circle).
   A node is ready to be executed as soon as each input receive a token.

3. the third proposal contains only nodes representing dataflow instructions and arcs between them showing data flow. See figure 2.2c for instance. Therefore, all operations like diversification of an output data (selecting output wires), controlling the loops and an access to shared memory are focused on the nodes. Also, it should be available to design a schme, where data flows always in a deterministic way. To satisfy these conditions, a node contains ports representing inputs and outputs and these ports are grouped to several portsets. One output and one input node can be connected by a wire. To ensure a determinsm, it is necessary to guarantee that there are not two parallel executions of the same node. Also, only one data token can be put to an arc, therefore an order of two consequential tokens can not be swapped.
   Data between two executions of a node can be shifted via arcs. The second possibility is to define two types of a node, which are different in storing data between particular executions. In Java language it means, that in first case the same instance of referenced class (the code of data-flow instruction) is used for each execution but it is guaranteed, that two executions can not be performed paralelly. In second case, always new instance of referenced class is created in each execution, therefore this soloution is the closest to pure data-flow node.
   A node is ready to be executed as soon as each input port of a certain portset receives a token.

### 2.2.2 Evaluation of the first proposal

**determinism:** there can be only one variable node from which arcs are coming out, but several variables, where a new value can be assigned to. Therefore, this solution is quite indeterministic since it's not guaranteed that two execution of a scheme will output the same result. For an instance, there are two data-flow nodes (A and B) which assign value to variable node (VAR) at different places. If both A and B nodes can be executed independent and paralelly, then in first case the node A can assign a value to the variable VAR before B, but in second case it can be assigned in different order.

**visual editor:** it is simple to implement, there are just nodes (processing nodes and variables) and arcs between them

**understanding of scheme:** is worse than in second solution since a scheme is not transparent due to „wireless" type of data flows.

### 2.2.3 Evaluation of the second proposal

**determinism:** these schemes are always deterministic since a processing inside a loop is separeted from the outside, a new iteration can start after the previous ended and on one wire, at most one token can be put.

**visual editor:** from the all proposals the scheme is most complex since it contains special structure for loops, cases and many other specialized entities implicated from language definition.

**understanding of scheme:** despite the scheme complexity and thanks to the properties ensuring determinism, schemes are good readable, e.g. it is clear where the flow is branching and which parts of a scheme represent a loop.

### 2.2.4 Evaluation of the third proposal

**determinism:** The scheme of proposal allows both deterministic and indeterministic data flows, and the variant depends on the engine implementation. A determinism is described in third proposal of the section 2.2.1, an indeterminism can be realized in the following way: at one wire can be put more than one token and several instances for that nodes can be created and executed parallely, which are not persisting data between iterations.

**visual editor:** simple to implement, there are only nodes and arcs between them

**understanding of scheme:** degree of understanding is in the middle of proposed solutions, because there can be several input portsets intended to start an execution and a loop need not be separated from from the rest of a scheme as in second proposal. However, no "wirelless" flow of data is present.

## 2.3 Selecting the most suitable solution

The best solution is supposed to be the most powerful with high degree of scalability which is especially necessary for running a program on a different hardware, to ensure a maximum power on dual cores computers as well as on computers with eight or more cores. Created schemes should be enough understandable and there should be an option if nondeterminism is preffered to create heavy parallel programs or it should not be allowed, especially when an order of processed data is important. From these prerequisities, the most suitable solution is third proposal which will be used for further analyses and it also will be implemented.

# Chapter 3

# The proposed components

The most interesting part on a scheme is the structure referring to concrete java code, it represents data-flow instruction and it is labeled as *processing entity* (PE). The example is shown at figure 3.1. The flow of data to and from PE is via *port*s that are grouped to *PortSet*s. Before a processing entity can be executed, all ports belonged to a certain portset have to contain received data (see figure 3.1, where are two input portsets, one contains InPort1 and InPort2, the second one contains InPort3 and InPort4, and at least one have to contain data to trigger execution of the entity). Similarly, only these ports will flow out data from PE via wires that belong to a certain output portset which is chosen in execution process(see figure, there are two output portsets, one is named as "Processed Flow", second as "Exception flow"). Thus one PE can have several input and several output portsets, input portset is chosen by data availability, the output in execution phase (in referenced Java code).

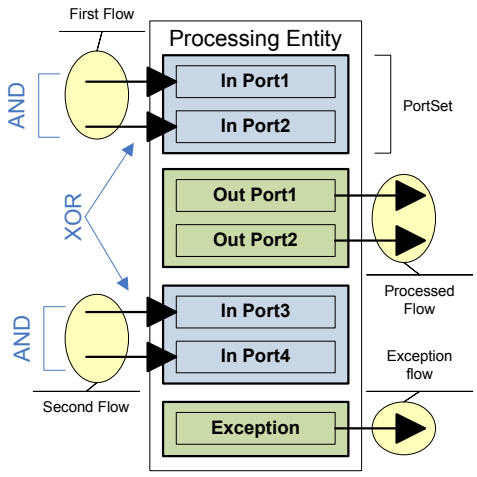



Figure 3.1: Data-flow instruction (processing entity), two independant flows as input, one as output

## 3.1 Loop and conditional flow (branching)

At the figure 3.2 is a scheme describing simple loop, first node is PE1 which represents „Loop operator" or „Loop driver" because all managing competences - begining and ending of a loop - there are hold by this node. As it was mentioned in previous section, the node can be executed if all ports in one of the input portsets have data available. Therefore, loop begins by receiving init data in „init" port. Process of PE1 execution is developed in java code where are produced resulted data and an output portset is selected for next run of program. It could be either portset with „data" port or „endOfLoop" port. If portset with "data" port is selected, then data flow is redirected thru PE2 and PE3 nodes. If "endOfLoop"

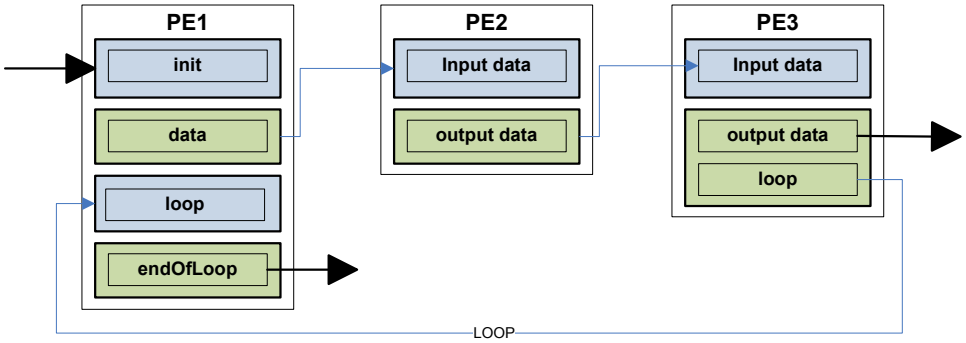



Figure 3.2: Loop example (blue color represents input portsets and green output portsets)

was selected, then loop processing is finished. PE2 node only process some data, PE3 node send signal to start new execution of PE1, but also ”output data” are send for furter processing.

## 3.2 Signal and data connection

Sometimes there are situations, when it is not necessary to send some data to another PE but only to send a signal that it is time to start a progress - e.g. to send new data loaded from database. Therefore, there were designed two types of tokens - data token and signal token. The main difference is that signal token do not carry a data needed for processing, but just to fullfil the port to get known for the engine that there is request for a new node execution. Also this signal is not visible in a java code - for both input and output ports. For example, a token between „loop” ports in figure 3.2 can be a signal, but this signal token will be helpful specially with a property introduced in the next section.

## 3.3 Persisting some information between two data-flow instructions

As described in section 2.1, the data-flow nodes should not leave any footprints after processing due to the possibility to be scheduled on the different processors, and also several instances of the same node can be executed in the same time. This condition is very important in increasing the effectivity of a program, although in some situations, like reading tokens from a file, it would be very useful to store some informations between particular executions. From this reason, two types of processing entities were proposed, Storage and Task. The difference is in persisting information after processing - node named *Storage* will be able to store information while node named *Task* not. In addition, all executions of Storage nodes will be thread safe and synchronised which means that in one time only one instance of a Storage can be executed - all demands for processing will be queued up.

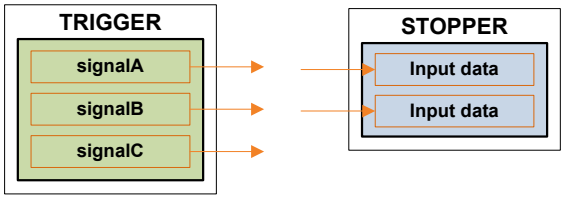



Figure 3.4: Trigger and Stopper componenets used to start and stop whole program

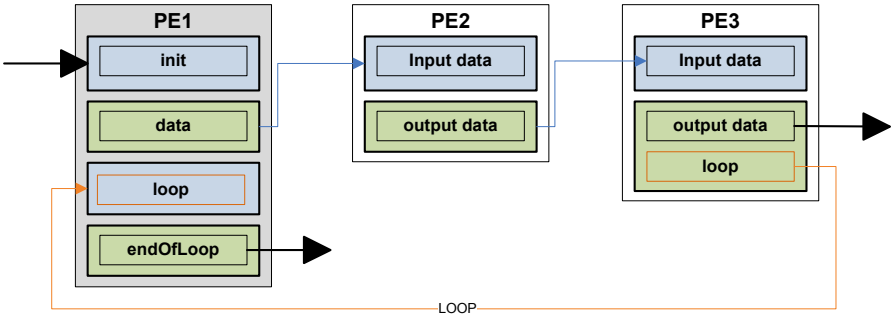



Figure 3.3: Loop example with Storage node(grey color) and signal token

At figure 3.3 is a loop example similar to one at figure 3.2, but here PE1 is a Storage (the gray background color). Also, PE1 does not need any data to start new execution, the execution is triggered by signal token(orange wire).

## 3.4 Trigerring first processing entities and ending the program

The whole program starts by executing ”TRIGGER” node, which has a special structure. It contains only output ports intended to send trigger signals to start executing of connected processing entities. No data ports can be placed here. Similar structure has ”STOPPER” component, but it is dedicated to receive signals. When an Stopper entity receives all required signals, then a program will end. If no Stopper is placed on a scheme, then program will end when no processing entity is active and no data or signals flows between them.

## 3.5 Structures overview table

| Type | Description | May contain [number] |
|---|---|---|
| Trigger | The whole program starts by executing this node. It contains only output ports intended to send trigger signals and start executing of connected processing entities. It can't receive any data and will be activated only once during program run. | Synchronization port [1..n] - output ports |
| Stopper | When this entity receives all necessary tokens, program will end. | Synchronization port [1..n] - input ports |
| Task processing entity | The execution unit which reference to a java class. When all data for a certain input portset are available, the execution process starts according the code developed java class. Before each execution, a new instance of this class is created. | Portset [1..n] - at least one input portset |
| Storage processing entity | The same as Task processing entity, except a new instance of referenced Java class is not created before each execution but only once. Therefore, it can store some information between several executions. | Portset [1..n]- at least one input portset |
| Portset | It is intended to group several ports which are required for a node execution. Also, for selected output portset, signals or data will be send from all its ports. | data / synchronization port [1..n], data port reference to java object |
| Data / Synchronization Port | It represents a place for a received token.<br>Data Port = for received JAVA object<br>Synchronization Port = for received signal, has state: received / unaccepted | - |
| Connection (also named as wire or arc) | It shows an connectiion between two ports of the same type (data - data, synchro - synchro). Each port can be connected to at most one connection. | - |

Table 3.2: Structures overview table

# Chapter 4

# Implementation

The implementation contains three parts: an implementation of a model based on scheme components (theirs definition is described in section 3 ), an implementation of an editor for the creation of scheme instances and an implementation of a runtime engine for a scheme execution. First, there was an analyse of suitable frameworks to be used for a model definition which will provide an easy way to design and also later, to make readable changes to a model. This framework should also provide a comprehensible API for working with a model, and it should provide an easy way to save created models to file in XML format. It is very important to have strong relationship of an editor and a scheme model, and since the project is a combination of a visual data-flow editor and a blocks of code developed in the textual java language, it would be very effective to have edit both parts in one multieditor program.

The Eclipse is an open development platform and it was selected as a best solution, which comprises the extensible frameworks, tools and runtimes for building, deploying and managing software across the development lifecycle. This platform contains frameworks for creation of the *Model-view-controller* architectural pattern with a comprehensive editor for a model design, and also it provides several tools for the code generation of a controller and custom views based on the designed model.

Graphical Modelling Framework (GMF)[13] provides a generative component and runtime infrastructure for developing graphical editors based on Eclipse Modelling Framework (EMF)[12] and Graphical Editing Framework (GEF)[14]. EMF will be used for model design, GEF for visual editor and GMF for easy transition from model definition thru controller to visual editor.

The last phase, and very important part, is an implemention of a runtime environment as a new eclipse plug-in.

## 4.1 Visual Editor

The power and the reason of using the GMF implicates from a possibility to create several meta-models which are related in a certain way to each other and consequently, on the base of this meta-models corresponding java code is generated using model to text framework (M2T)[16]. Finally, generated code is customized to cover special requirements of the proposed editor. In addition, the M2T generators can be reused to handle changes in meta-models after customization of

a final code, but changes made in the first approach will not be overwritten. In the following section, important properties of some meta-models are described. The list of meta-models and also the process of an integration of new plugins into Eclipse platform are demonstrated at figure 4.1.

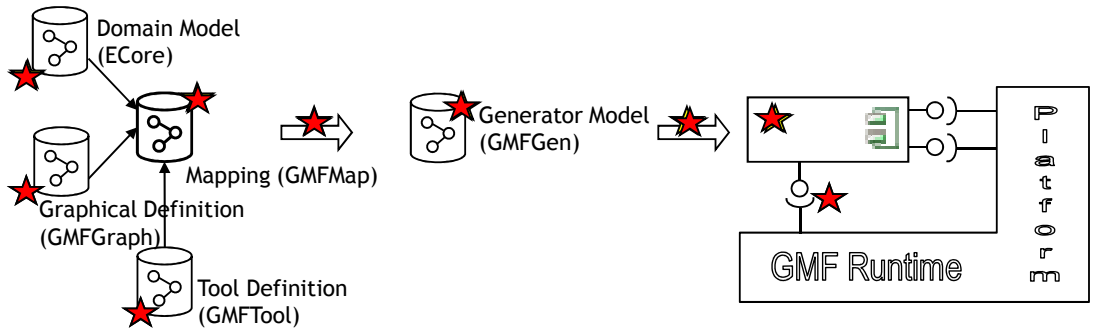



Figure 4.1: Process of creating MVC architecture by GMF and its integration to Eclipse

In the result, four plugins to Eclipse platform represents MVC architecture:

**dataflowEditor** represents domain model part but also all meta-models of GMF are stored here

**dataflowEditor.edit** represents controller

**dataflowEditor.editor** represents simple non diagram editor but it's possible to modify the underlying model

**dataflowEditor.diagram** represents graphical visual diagram editor, implements the proposal's structures

### 4.1.1 Domain Model

This file represents a domain model based on the proposal, it is named „model.ecore" and its diagram is shown at figure 4.2.The virtual top structure is named „Block", its instance is created in the behind of user actions because it is the root element. This root contains other elements therefore it is only one that can not be put on the scheme. The other structures can be placed by users, *Block* element may contain: exactly one *Trigger*, several *Stopper*s, several processing entities (PE) which could be *Task* or *Storage*, and some *Data* and *Synchro connections*. For all these objects Block element represents theirs containment. Trigger element can contain only synchronization ports, therefore the PortSet element is omitted here. The same is valid for Stopper. The majority of entities attributtes have „EString" or „EInt" types (String and int in Java), but there is also special enum „ConnectionType" intended to assign the orientation of ports and portsets, and custom type „SourceType" which in fact it represents EString.

On the base of the „model.ecore" file, a „model.genmodel" meta-model was generated where it is possible to customize some controller and simple editor's properties. From „model.genmodel" file these plugins are created **dataflowEditor**, **dataflowEditor.edit** and **dataflowEditor.editor**.

Instances of domain model are saved in XML format.

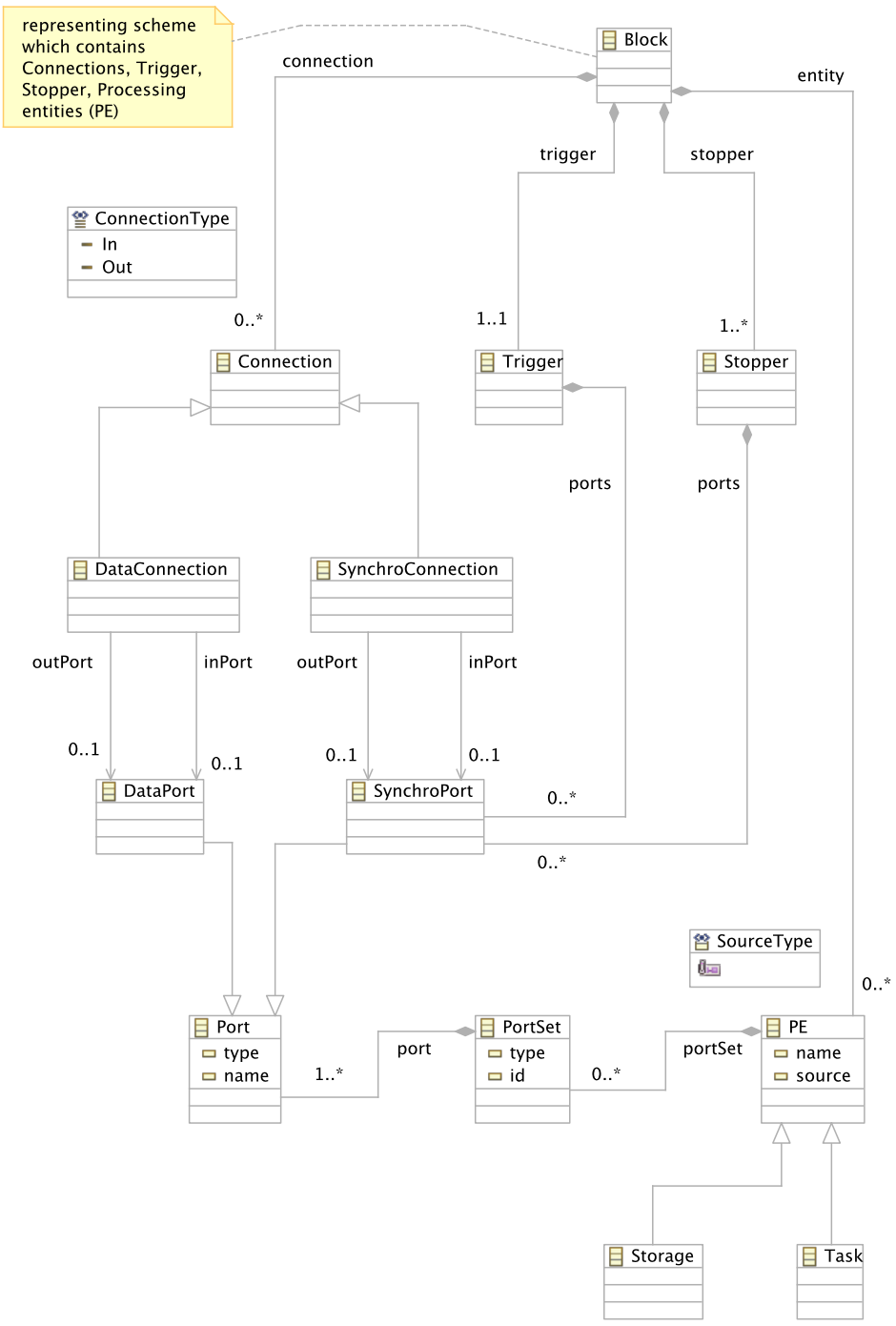



Figure 4.2: EMF domain meta-model

### 4.1.2 Diagram meta-models creation phase

The source of the diagram plugin is generated at the base of the following meta-models:

**model.gmfgraph** meta-model which defines graphical figures by describing nodes, labels, containments, lines, relationships between them, resize policies, layouts and some other properties. For instance, Trigger is defined as a node with special figure descriptor containing a label and a rectangle for ports with a stack layout, supplied by an access from an API.

**model.gmftool** here is defined pallete of tools for creating figures on a scheme (Trigger, Stopper, Task, Storage, DataPort, SynchroPort, PortSet, Synchronization Conn. and Data Conn.), but also there is the place where some toolbars, main, popup or context menus and custom icons for tools can be defined. For this project, only pallete tools were proposed.

**model.gmfmap** here is the place to explane how several meta-models are related to each other. For an instance, here is a definition how a tool creating a certain figure is maped to a certain element from domain model. Mapping contains canvas (maps to a Block element from a domain model), links(synchro, data) and a top node references(Storage, Task, Trigger, Stopper) which have structures similar to that in figures descriptors from gmfgraph, except that there are mappings to tools and model elements.

**model.gmfgen** like genmodel to ecore, this is a semi step for a code generation of a diagram code. Many properties can be set here, e.g. class names, option to use a live validation while creating schemes, file extensions or using a navigator.

Some customizations have been done in the resulted generated code (some special layouts, custom anchors for wires, rectangular property of wires, ...) and the list of changes is delivered in the included CD.

## 4.2 Run-time Environment

This section is divided in two parts, first describes the structure and meaning of particular parts of execution engine, the second part porting this engine to Eclipse as a new launcher.

### 4.2.1 Execution Engine

Engine is implemented as pure Java standalone application, main class is *dataflowscheme.engine.core.Runner*. It takes several parameters, see section 5.3.3. The whole engine structure can be divided according threads used to handle specific parts, which are:

1. Main thread - handle main class *dataflowscheme.engine.core.Runner*. It takes care of initialization where data-flow graph is build from loaded XML scheme. It starts the TRIGGER entity and wait until program will finish.

2. Event Parser thread - this thread takes care of sending data between processing entities and generating execution events after an entity is ready for execution (for certain input portset, all tokens are available)

3. Execution Event Parser thread - this thread takes care of executing processing entities. For its execution, it has thread pool at disposal, which it is used to schedule each event for separate thread.

4. Activity Checker thread - this thread is checking an activity of whole program, and if no STOPPERs entities are present in scheme or some deadlock occurs, then it will recognise this state, and stop the whole program correctly.

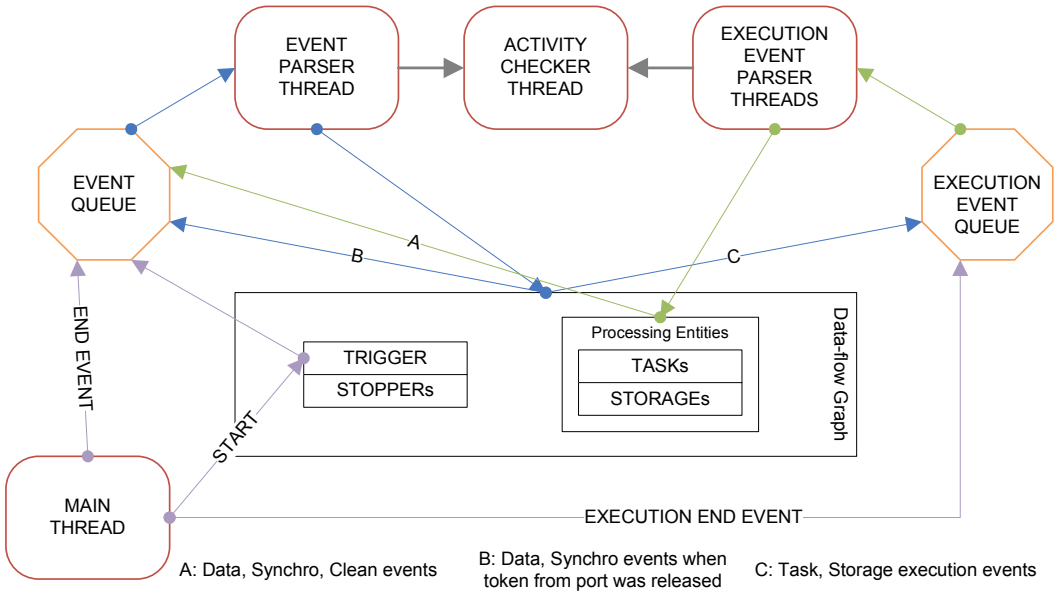



Figure 4.3: Overview of the Engine structure

### Main thread

Here is the list of most important steps done in Runner class:

1. First, both event parsers are created and there are started in separate threads. Each parser is taking events from appropriate queue and then the event is handled according its type:

   (a) Event Queue in duo with Event Parser: deliver signal and data tokens to processing entities (Task, Storage), but also it handles clean event (to invoke execution of entity, if it has no output tokens to be send) and end event for parser (which is created when program is ending)

   (b) Execution Event Queue in duo with Execution Event Parser: have at disposal thread pool for submiting requests about Task and Storage execution. End event is to stop parser correctly.

2. At the base of data-flow scheme, which is loaded from XML file, a data-flow graph is build (see Runner.buildDataflowGraph()):

(a) Loading - a scheme (an ecore domain model instance) is loaded from XML resource by EMF framework's libraries. The structure of objects and relations between them corresponds to domain meta-model shown at figure 4.2. For instance, the class of ecore object for TASK is *dataflowScheme.Task*. (see Runner.loadResourceFromFile())

(b) the ecore objects like Task, Storage, Trigger and Stopper are encapsulated by corresponding engine core objects (e.g. for Task it is CTask) to be able to receive and send events, to be executed in case of TASK and STORAGE, and to handle other engine requirements. Here is also an initialization of all core objects, for CTask and CStorage the initialization means to check an integrity of referenced source with model definition, there is a list of steps for CStorage initialization (see Runner.initProcessingEntities()):

i. check that referenced class was put to the classpath, because all this class is loaded to JVM in the runtime (via Java Reflection API)

ii. check that referenced class contain *run* method, which it is invoked in each execution. This method has to be public, take one Integer parameter and return Integer

iii. check that referenced class contains methods for data ports. In case of input port, method has to have name of *setPORTNAME* where PORTNAME is the name of the port from loaded scheme. Also it has to take one parameter of type java.lang.Object. In case of output port, particular method has not parameters, its name have to be *getPORTNAME* and must return java.lang.Object. In addition, it is checked, that there are not two data ports with the same name.

iv. for CStorage entity it is checked, that referenced class contain *start* and *stop* methods, which does not take any parameters and doen not return any value.

(c) from connection between entities (Task, Storage, Trigger, Stopper) the graph is build with Trigger entity as root node (see Runner. buildDataflowGraphFromConnections())

3. Activity Checker thread is started

4. Now, cTrigger is invoked, where signal tokens are send to all entities connected with Trigger. To send token, first appropriate event is created (Data or Signal event) and then it is put to event queue (see figure 4.3). These events are parsed by event parser thread.

5. Main thread is waiting until program will end. As synchronization aid, it is used *java.util.concurrent.CountDownLatch* as one signal countdown to wait until one of Stopper entities will decrease it, when main thread will wake up. Also an Activity Checker can decrement the count of the latchwhich is done when deadlock occur or when there are no more tokens to be send.

6. Main thread was awoken, therefore end events for both parsers are queued and main thread will wait after both parsers are shutdown.

7. For all CStorage entities, a finalization of referenced classes is invoked (*stop* methods are called on referenced classes). Invoking special methods on CStorage reference classes when program is ending can be used, for instance, to close connection to database or to close opened files (see StorageSourcesHandler.finalizeStorageInstances()).

8. Since no more signal or data events can be send, and no execution can be performed now, it is checked for all unused data which were produced but not send, and for all data and signals, which were put to ports, but they were never used (see Runner.processUnusedData()).

9. In addition, some simple but usefull statistics for program profiling can be print (if were enabled when program starts) like (see ExecutionEPstats.printStats()):

   (a) how much time takes initialization phase of program (steps 1-3) and run-time phase (steps 4-5)

   (b) how much processing time takes engine service - sending, receiving, rejecting events and processing data produced by referenced classes. All the rest time was used for creation of referenced classes and invoking its *run* methods. A service rate information can also provide an information about grain structure of data-flow program. For instance, a service rate of 70%[1] means that program is too fine-grained and in the scheme there are many entities with too simple code in referenced classes.

   (c) how much processing time of all execution event parser's threads was used for particulat processing entities (STORAGE, TASK)

   (d) what was the load factor of event parser - rate of time while thread was suspended due waiting for events and while thread was busy with engine service. When a rate reachs 100%, then it means that program can not run faster and the boundary was reached.

**Event Parser thread**

Event Parser thread is processing the Event queue (see figure 4.3) - in each iteration it takes one event (or thread waits until an event is available) and invoke its processing dependent on type of an event . The processing of events encloses these actions:

1. send signal or data token from output entity (where token was produced) to target entity (see PE.receiveData(), PE.receiveSignal())

   (a) check, if a port in target entity does not contain a token. If it already contains a token, then mark an unsuccesful attempt for certain port and discard en event. Otherwise, set a token to particular port.

   (b) check, if a portset has all required tokens (data and signal). If it has, then there are two possibilities:

[1] 70% was reached in data-flow program where data flows between three entities in 10 000 iterations. The code of entities was very simple, just a concatenation of two String objects

i. if previous execution of particular Task or Storage is finished yet, then create new Task or Storage execution event

ii. if an execution is still in progess then queue a demand for execution. Each Task or Storage has own queue for execution event (but for Task and Storage execution events is only one global queue - Execution Event queue)

(c) Since a token was put to a port successfully, then send a response to an opposite output port (where a token was send from), see PE.receiveResponse() with message type "delivered_ok". A token is remove from a list of produced data or signals intended to be send. If all data were sent (the mentioned list is empty) then following steps are performed:

i. cleaning before new execution (see PE.cleanBeforeNewExecution()) - set flag that execution ended, delete all data from portset used for execution and send a response about "free port" in case that some tokens were rejected (these tokens are send again).

ii. since a Task or Storage is ready for new execution, it is checked if there are some demands for execution saved in queue. If there are, then first demand is pull out and an action from (b)i. is performed.

2. send "clean" event to an entity. This event is produced in one of Execution Event parser threads when an entity after execution has not any data to be send (was not selected an output portset) and therefore action 1.(c)i. can not be performed (Task or Storage is cleaned after all data and signals from selected portset were sent). The reason of "clean" event is to invoke cleaning manually (action 1.(c)i.), but not in execution parser thread but in event parser thread (no locks are necessary). See CleanEvent.execute().

3. handle event parser end event - this action will ensure, that event parser will not wait for new events anymore and this thread will be end up correctly.

**Execution Event Parser thread**

Execution Event Parser thread is processing the Execution Event queue (see figure 4.3) - in each iteration it takes one event (or thread waits until an event is available) and invoke its processing dependent on type of an event . The processing of each event is performed in separate thread and it encloses these actions:

1. Task execution event

(a) invoking part (see TaskExecutionEvent.invoke()) - new instance of referenced class is created and for all data ports from particular portset are called "set" methods on a newly created instance. Consequently, a "run" method of an instance is invoked, which returns the number of output portset or null, if no data and signals tokens should be send.

(b) processing of produced data (see PE.receiveResult()) - on the base of portset number returned by "run" method from previous step, for all

ports of selected output portset, data are fetched (see PE.parseResult()). Afterwards, either data and signal tokens are send (new data and signal events are created and put to event parser queue) or a "clean" event is created to clean all data used for execution from particular input portset.

2. Storage execution event - process of this event is similar to process of Task execution event. The main difference is in creation of new instances, where for one Storage only one instance during program lifecycle is created, but for Task, new instance is created in each execution event. Since the same storage can be put on the scheme at several places, the access to referenced instance is synchronised - it is secured by synchronization lock (there can not be two parallel executions of the same storage).

3. Executor End Event - this event ensures, that execution event parser will not wait for new events anymore and this thread will be end up correctly.

**Activity Checker thread**

The reason of the Activity Checker is to control the state of the program and if there are not more events to be processed then this state will be recognized. This situation can happen in three cases:

1. a stopper decrease the synchronization latch (see section Main Thread, point 5) and program ends up. Therefore, it is not necessary to check the program state anymore.

2. input scheme has no STOPPER entity and a program can not end as in the first case. Therefore, when no activity will be recognized, Activity Checker will decrease the synchronization latch to end up a program.

3. input scheme can have some stopper entities, however if required signals to at least one stopper were not delivered (e.g. due to wrong design of a scheme) then a stopeer will never decrease the synchronization latch. It is a dead lock case. Activity checker can this state recognize and end up a program as in the second case.

The idea of the activity checker is implementated in the following way: each second it is checked, if there are some unfinished execution events (Task or Storage) and also if the event parser is processing an event. If both conditions are negatives, then the state of a program is marked as inactive and the synchronization latch is decreased. The minimum time to recognize an inactivity are two seconds, because the activity checker begins with one second delay, afterwards, if there is not activity because both conditions are negative, then a verification is made in a next second.

### 4.2.2 Conclusion about implementation of the engine

The main advantage of the implemented engine is an event system. Even if the engine is multithreaded, it was not necessary to use synchronization locks in access to data, because only one thread - event parser thread - has an access

(locks are used only in one case to ensure that an instance of a referenced class of a certain storage is not executed parallely). All the other threads - threads of execution event parser - are only processing data received to portsets, and after an execution, these threads are generating some events, which are consequently processed by the event parser. A disadvantage of an event system could be high rate of the engine service when a program structure is too fine-grained and in the scheme there are many entities with too simple code in referenced classes. Also, this implemented engine can have a bigger memory consumption as if a program was implemented with a custom simple producer-consumer design pattern. It is caused due to an existence of references to ecore classes (which are used to load a scheme) during whole lifecycle of a program. A better memory consumption is a case of a future improvement of the engine.

### 4.2.3 Implementation of custom Eclipse launcher for data-flows programs

Custom Eclipse Launcher for data-flow programs is composed from two plugins: *dataflowscheme.engine.ui* and *dataflowscheme.engine.core*. It is implemented as an extension of common java application launcher, although there are some differences:

1. UI plugin: instead of JavaMainTab(it can be seen in a Java application launcher), there is defined DFSMainTab to specify a data-flow scheme and some engine properties . All necessary parameters from launch configuration are dispatched to a delegate in a core plugin via properties. Names of custom properties are defined in IDFConstants class.

2. Core plugin: there is implemented a delegate to run a java application in a background. If a data-flow program was started in a debug mode, this delegate will ensure that an Eclipse Java Debugger will be used (there is possible to set breakpoints in an editor of java classes and subsequently a program execution will stop there). The implemented engine is a standalone java application with a *dataflowscheme.engine.core.Runner* main class. On the classpath, there are required libraries like *org.eclipse.emf.common*, *org.eclipse.emf.ecore.xmi*, *org.eclipse.emf.ecore* which are part of the EMF project, but also the binaries of a *dataflowEditor* and *dataflowEditor.edit* plugins. All these libraries are necessary to load a data-flow scheme from a XML file. Since the engine loads referenced classes of Task and Storage entities to JVM dynamically (Java API reflexion), these classes have to put to the classpath as well.

# Chapter 5

# User Guide

The data-flow editor and the execution engine are intended to create and execute data-flow schemes. This document describes how to obtain, install and use these tools to create and run data-flow programs.

## 5.1 Requirements

### 5.1.1 Java version

Eclipse is a Java program available for various platforms: Windows, MacOS, Linux 32 and 64bit. To run the Eclipse[11], it is necessary to have Java installed, many Java Virtual Machines are supported, but the most commonly used is Java from Sun Development Network[17]. Plug-ins for Eclipse provided by this project were developed by Sun's Java Development Kit version 5, but also Java version 6 can be used to run and compile created programs.

### 5.1.2 Eclipse version and required plug-ins

Editor and execution engine were developed and tested on:

- Eclipse version 3.4.2[11]
- Eclipse Modelling Framework 2.4.0[12]
- Graphical Modelling Framework 2.1.1[13]
- Graphical Editing Framework 3.4.2[14]

The Eclipse platform and all required plug-ins can be found at Eclipse Project website[15].

## 5.2 Installation

The prerequisity is to have installed requirements from previous section. A binary package contains four plug-ins intended for scheme definition and editor

- dataflowEditor.jar

- dataflowEditor.edit.jar
- dataflowEditor.editor.jar
- dataflowEditor.diagram.jar

and another two plug-ins are necessary for program execution

- dataflowscheme.engine.core.jar
- dataflowscheme.engine.ui.jar

For these plug-ins, it is necessary to copy them to the *plugins* directory of the Eclipse program. Then run the eclipse with a *clean* mode by command

```
eclipse -clean
```

## 5.3 Usage

The delivered plugins provide the new wizard for a model diagram creation, the specialized editor for diagrams, the simple editor to edit a domain model and the new launch configuration to execute diagram schemes in both run and debug modes. The debug mode here means to use eclipse JVM Debugger to debug referenced blocks of codes. In the sections 5.3.2, 5.3.2 and 5.3.2 there are described steps how to create a new dataflow project and main processing entities Storage and Task with a referenced code.

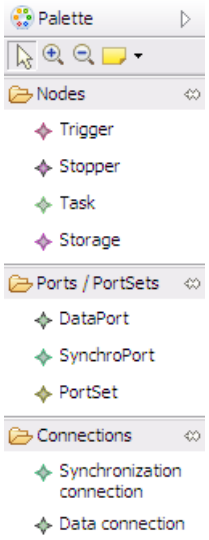



Figure 5.1: Palette

### 5.3.1 The Palette, Properties View and Description of elements

The palette is intended to be used for adding elements to the scheme, for zooming and attaching notes. Each element instance on the scheme has some properties which are accessible in *Properties View*. To show this view, click with a right mouse button on an element instance and select „Show Properties View". Usually two tabs are visible, the Core tab with specific properties for an element and the Appeareance tab to edit visual style, e.g. to change a text style. Here is the enumeration of special features of some elements from the palette:

**Task and Storage:** on the Core tab in Properties view, the name and referenced java class can be set here. If source is set, then referenced java class in java editor can be opened by double click on the header of Task or Storage element.

**DataPort:** the visual feature is a black square and letter „D" next to the port name. The orientation of a port is illustrated by rendered square: on the left it means input type, on the right output. The orientation can be changed on the Core tab.

**SynchroPort:** it is similar to the DataPort, but the orientation is rendered by a circle and port type is marked with letter „S"

**Data Connection:** it is displayed as a blue line and it has always a rectilinear routing style.

**Synchro Connection:** it is similar to the data connection, but has orange color.

### 5.3.2 Developing the data-flow program

Following three procedures indicate a way, how to establish new project and consequently there are described creations of the Task and Storage processing entities, which are part of examples delivered on the CD, see appendix C. The screenshot of an example is shown at figure C.1.

#### Creation of data-flow project

To create new a data-flow project with empty diagram model, the following steps are required:

1. Create a new *Java Project* by selecting „File -> New -> Java Project" in the Menu bar
2. Set a project name to *DataflowProgram* and press „Finish"
3. Select a DataflowProgram project in *Package Explorer* and create a new Folder named *models* by „File -> New ->Folder" and press „Finish"
4. Select a models folder and create a new data-flow diagram by „File -> New ->Other" and find a wizard named „Model Diagram", press „Next"
5. Choose a name for *diagram model* (e.g. example1.dataflowscheme_diagram) and for a *domain model* (e.g. „example1.dataflowscheme"). Press Finish.

#### Creation of Task processing entity

In the following steps there is described how to create a data-flow instruction of the Task type. The goal of this task is to determine if a number is prime or not.

1. Select *Task* in Palette, put it on the scheme and set a name to *PrimeDetector*
2. Select *PortSet* in the Palette and put it in Task in the place where mouse pointer has cross for an addition.
3. Select *DataPort* in the Palette and put it in the PortSet created in the previous step, set the name to number.

4. Create a new PortSet as in step 2, and two data ports inside. Named them as *outNumber* and *resolution*.

5. Set the orientation of last two data ports to *Out*.

6. Select a src folder in Package Explorer and create new class via menu. Set a name to PrimeDetector and package name e.g. to „dataflowProgram.example1".

7. See the referenced java source code in Appendix B.1 and paste it to a newly created class.

8. Show the properties of Task and find the referenced class via *File Selection* dialog opened by click on three dots in Source row of the Core tab. The result should be „/DataflowProgram/src/dataflowProgram/example1/PrimeDetector.java"

**Creation of Storage processing entity**

The principle of a Storage creation and required steps are the same as for the Task entity described in previous section. The example source code for a storage *SavePrimes* is attached in the appendix B.2.

### 5.3.3 Execution of program

A dataflow application can be run via custom Eclipse launcher by creating a new launch configuration in left panel (see figure 5.2). Then it is necessary to set a dataflow scheme and name for a new launch configuration. It is also possible to set some engine properties, the level of logging (Debug, Info, Warn, Error, Off), the default value is Info which is also recommended. As default log4j[10] configuration, a log is printed to console. However, it is possible to create custom configuration file, and set tell to engine to use it (option *Use custom log4j properties file*). If *Compute and print Statistics* option is checked, then some simple but usefull statistics for a program profiling will be print :

1. how much time takes initialization phase of program and run-time phase

2. how much processing time takes engine service - sending, receiving, rejecting events and processing data produced by referenced classes. All the rest time was used for creation of referenced classes and invoking its *run* methods. A service rate information can also provide an information about grain structure of data-flow program. For instance, a service rate of 70%[1] means that program is too fine-grained and in the scheme there are many entities with too simple code in referenced classes.

3. how much processing time of all execution event parser's threads was used for particulat processing entities (STORAGE, TASK)

[1] 70% was reached in data-flow program where data flows between three entities in 10 000 iterations. The code of entities was very simple, just a concatenation of two String objects

4. what was the load factor of event parser - rate of time while thread was suspended due waiting for events and while thread was busy with engine service. When a rate reachs 100%, then it means that program can not run faster and the boundary was reached.

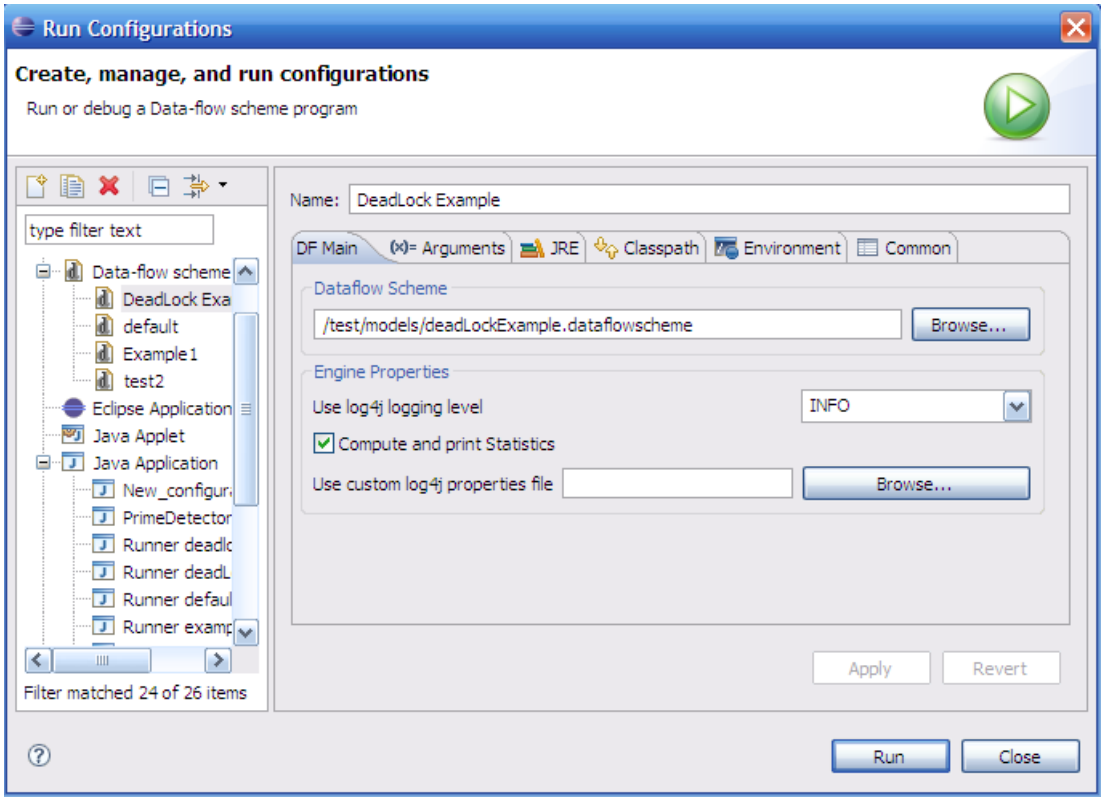



Figure 5.2: Eclipse Launcher for a dataflow program

# Chapter 6

# Related Projects

## 6.1 NI Labview

LabVIEW[8] is a commercial data-flow visual programming language developed by the National Instruments company since 1986 and according to [8] it is used by millions of engineers and scientists. Primary it was intended to be used for data analysing in laboratories by non profesional programmers, which implicates well-arranged and fine grained design of data-flow language. Program is graphically represented as scientific instruments with various controls like buttons, LED indicators, switches, listboxed and many others, which in fact it makes an impression as a real instrument. This instrument is implemented in a block diagram, where controls are indicated as variables, and the whole program is developed by embedding various predefined data-flow instructions, loop or case blocks to the scheme and by connecting them with wires. The structure of the Labview language is fine grained, where, for instance, to sum two numbers, it is necessary to add PLUS instruction to the scheme, connect two wires as input and one as output. Some information about loop definition, preserving deterministic process of program computation can be found in section 2.2.

## 6.2 Pervasive DataRush

Pervasive DataRush[13] is a commercial, complex Java framework intended for developing a data-intensive application and to ensure high dynamical scalability on a different hardware. Moreover, all parallel programming complexity is hidden in delivered execution enviroment library. The data-flow idea in DataRush is expanded in following level: data-flow instructions represents graph's nodes, there are connected by oriented edges via predefined ports, and these oriented graphs are acyclic. To ensure flow of data, Data on some nodes are subsequently generated and put by one to the output, from where data token flows along edge to connected node's input. The node is fired when all incoming data are available. If a node has not inputs, it is only generating data and time of interupts between iterations is handled by engine. There is no visual editor for creating data-flow schemes, nodes are implemented by extending classes from delivered library. Definition of inputs and outputs can be specified by a definition in XML format or by using special syntax in java class constructors. Here is the example of simple loop used for data generation:

**Algorithm 6.1** Simple lopp used for data generation

```
while (y.stepNext()) {
  if (x >= y.asInt()) {
 z.push(x);
  } else {
     z.push(y.asInt());
  }
}
```

Synchronization is implicit in the stepNext() and push() methods of input and output ports. The power of this framework is in heavy parallel engine ensuring high dynamical, horizontal and vertical scalability according of hardware capabilities. Moreover, the execution environment performs thread monitoring and dataflow deadlock detection so that the Java developer does not need to undertake complex, specialized Java parallelizing development tasks.

## 6.3 Comparision with suggested solution

Even both projects impelement data-flow paradigm, there are intended for different purposes. The first project is focused on a programming of instruments to communicate with specialized hardware by composing of predefined structures visually. Such programs can be very easily and in nice way debugged in Labview enviroment, however, due to some restrictions of loops structures and due to ensure determinism, this framework is not suitable for heavy parallel application. On the other hand, DataRush does not share some Labview advantages, altought as a development language it is used widely spread Java language. Developing programs with DataRush will result in creation of very powerfull applications.

The goal of this project was to combine advantageos properties of both Labview and DataRush projects in a meaning, that a data-flow structures are created visually in an editor and saved to XML but instructions as well as an engine core are implemented in Java. Since DataRush use both Java framework and XML files for data-flow scheme creation, its editing in graphical editor would be quite difficult, especially to ensure proper propagation of changes in both parts. Therefore, in this project, Java and data-flow parts are clearly divided, so creating schemes in an editor has not direct impact to referenced Java code.

In this project, it was necessary to introduce portsets due to requirement to show loops visually. The reason is, that an execution of an instruction is bounded by a method of referenced class and the only way how to call a new iteration is to call the method again. However, it is necessary to quit the method to forward produced data to connected nodes (in DataRush, there could be while loop and produced data can be flowed out from an instruction inside the loop, see code in section 6.2).

Even this project does not contain many amazing features from both products, it was designed with aspects for further extensions, for instance it allows creation

of visual debugger similar in Labview or to extend an engine to increase the power of whole program computation.

# Chapter 7

# Conclusion

The main goal of the thesis was to analyse a possible combination of data-flow concept and Java language in the meaning, that a developing phase of a data-flow part of a program is to accomplish it via graphical editor but data-flow instructions are coded in Java language and all work with running a program and execution of instrutions with emphasis to proper threads operating is done in engine core.

In the begining of thesis I dealt with the data-flow features and its applying to Java language, where I identified some inconsistences mainly in using of a static memory. This led to the restrictions of using static variables, except final primitive data types which are considered to be non changeable constants. From the rest non conflict features like data dependencies equivalent to scheduling I have based the analyse which I handled with orientation to solve the open question of a loop representation in visual data-flow languages. There I outlined several proposals with an intention to compare them in general lucidity and its understanding, severity of an editor implementation, the capability of an indeterminism occurence and possible future extensibility in an efficiency issue.

From the results of the analyse, I identified the required properties of particular structures. Among them there belong two types of processing entities different in persisting of data between several executions. The first processing entity is proposed to not persist any information which it is dedicated from data-flow properties and the reason is to guarantee independent order of an instruction execution. As a future extension, the instruction can be executed in a separate memory, e.g. on the remote computer. The reason of introducing second type of processing entity, which persist information between execution calls, is to simplify the scheme e.g. in case of reading from file or database, where are expecting several executions with only data producing purpose. In addition, all executions of the same entity are in the second case sychronized.
The rest of proposed structures are ports given for representing inputs and outputs of processing entities, and portsets, which are dedicated to group ports needed and sufficient for execution of processing entities that represent data-flow instructions. To clarify reason of portsets, their definition makes possible to flow out data only to certain ports as well as to execute data from different, independant flows, usefull in loop controlling (e.g. to separate initialization phase and shifting data among iterations).

As the verification of analyse result, there were implemented graphical editor

for a scheme creation and the execution engine to operate them. The implementation was succesful, although there are some issues for a future extensibility.
The first very usefull addition could be an extension of debug mode in a way of reflecting the activity of processing entities and also to display flows of data. The debug mode would allow to set breakpoints for data-flow intructions and to trace them where one step can represent sending data over an arcs from all entities which finished its execution. Currently it is possible to use only Eclipse java debugger, to implement the proposed extension it would be necessary to extend the editor and the engine core. A proof, that this extension is possible to implement, was already done.
The second addition could be an extension of the engine to improve the scalability of a program. Currently, there can not be several parallel executions for one processing entity and in a port, there can not be stored more than one data token. The idea is to allow both restrictions which can lead to an indeterminism, but from the other side, the overall scalability at different powerfull hardwares can be much more improved. One of the disadvantages can be a difficulty to avoid a creation of many data tokens for one port, but an advantage can be a possibility to distribute an execution power dynamically to the most time consuming processing entities.

As a lack of current state of an editor implementation it can be considered a relationship in development phase between processing entity structure and referenced code (e.g. automatic generation of a code according the entity definition in a schme ), missing of specialized structures (e.g. to join several wires) but also the necessity to fill names for synchronization ports.

# Bibliography


[1] Johnston W. M. , Hanna J. R. P., Millar R. J. (2004): Advances in Dataflow Programming Languages. ACM Computing Surveys (CSUR), v.36 n.1, p.1-34

[2] Baroth E., Hartsough Ch. (1995): Visual programming in the real world. In Visual Object-Oriented Programming: Concepts and Environments. Prentice-Hall, Upper Saddle River, NJ, 21–42

[3] Bitter R., Mohiuddin T., Nawrocki M. (2001): Labview advanced programming techniques, CRC Press, Boca Raton

[4] Bruce Eckel (2002): Thinking in Java, Prentice Hall Professional Technical Reference

[5] Johnson G. W. , Jennings R. (2001): LabVIEW Graphical Programming, McGraw-Hill Professional Publishing

[6] Almasi G. S., Gottlieb A. (1989): Highly Parallel Computing, Benjamin-Cummings publishers, Redwood city, CA

[7] Labview Loop and While Loop Structures
`http://zone.ni.com/reference/en-XX/help/371361B-01/lvconcepts/for_loop_and_while_loop_structures/`

[8] NI Labview home page
`http://www.ni.com/labview/`

[9] Pervasive DataRush home page
`http://www.pervasivedatarush.com`

[10] Home page of Apache logging services
`http://logging.apache.org/log4j/`

[11] Eclipse Application home page
`http://www.eclipse.org`

[12] Eclipse Modelling Framework home page
`http://www.eclipse.org/modeling/emf/`

[13] Eclipse Graphical Modeling Framewrok home page
`http://www.eclipse.org/modeling/gmf/`

[14] Eclipse Graphical Editing framework home page
`http://www.eclipse.org/gef/`

[15] Eclipse download page with several distributions for various platforms
http://www.eclipse.org/downloads/

[16] Eclipse Model to Text project home page
http://www.eclipse.org/modeling/m2t/

[17] Sun Developing Network home page
http://java.sun.com/

[18] Java API, version 1.5
http://java.sun.com/j2se/1.5.0/docs/api/

# Appendix A

# Content of attached CD

The enclosed CD contains:

- Eclipse application for MS Windows operating system with all required plugins (with implemented plugins)
- Eclipse application for Linux operating system (32bit) with all required plugins (with implemented plugins)
- Workspace for Eclipse application with implemented plugins (sources)
- Workspace for Eclipse application with delivered examples and configurations
- User Documentation
- List of customized classes for the generated visual editor

# Appendix B

# Sources of processing entites from User Guide

**Algorithm B.1** Referenced java class of Task *PrimeDetector*

```
public class PrimeDetector {
private Object input;
    private Object resolution;

    public Integer run(Integer portSet){
        if(input != null){
            Boolean res = null;
            res = rabinMillerComp((Double)input);
            resolution = res;
        }
        return 2;
    }

    public void setInNumber(Object o){
        this.input=o;
    }

    public Object getOutNumber(){
        return input;
    }

    public Object getResolution(){
        return resolution;
    }
}
```

**Algorithm B.2** Referenced java class of Storage *SavePrimes*

```
public class SavePrimes {
    private Object number;
    private Object resolution;

    public Integer run(Integer portSet){
        if(number == null) {
           return 2;
        }
        if((Boolean)resolution == true){
           System.out.println("Number " + (Double)number
+ " is a prime.");
        }
        return null;
    }
    public void setNumber(Object o){
        number = o;
    }

    public void setResolution(Object o){
        resolution = o;
    }

    public void start(){}
    public void stop(){}
}
```

# Appendix C

# Description of delivered examples

## C.1 Determination of prime numbers

This example describes a determination if some numbers are composites or primes. All structures are used, Trigger, Tasks, Storages and Stopper. Storage Generator is generating numbers, in each execution it generates two numbers for two Tasks - Prime detectors. Task Semaphor is only used to put results from detectors to Storage SavePrimes, where if a prime was found, then it will be printed to the console (or it can be saved to file). When the Generator send null references thru Tasks to SavePrimes storage, then a signal to the Stopper is send and the program ends. See figure C.1.

The name of the scheme file is *example1.dataflowscheme_diagram*.

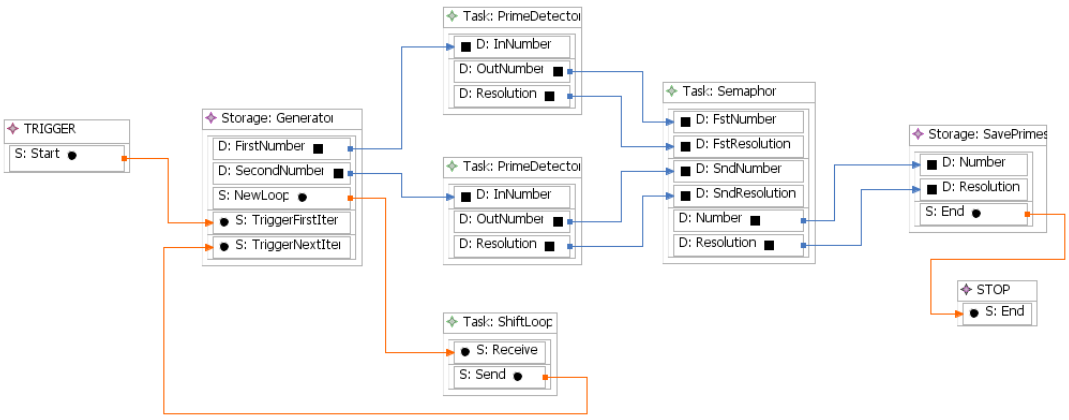



Figure C.1: Selecting prime numbers

## C.2 Extended determination of prime numbers

Differences from previous example are:

1. no Stopper is present, therefore program ends when Activity Checker thread recognize a program inactivity.

2. Although there are two Generator storages and four SavePrimes storages, in runtime for a whole lifecycle of a program, only one instance of Generator

ans SavePrimes class is created, and access to these instances is thread safe. Program should run faster than previous example on four core processors.

3. the name of the scheme file is *example1Extended.dataflowscheme_diagram.*

See figure C.2.

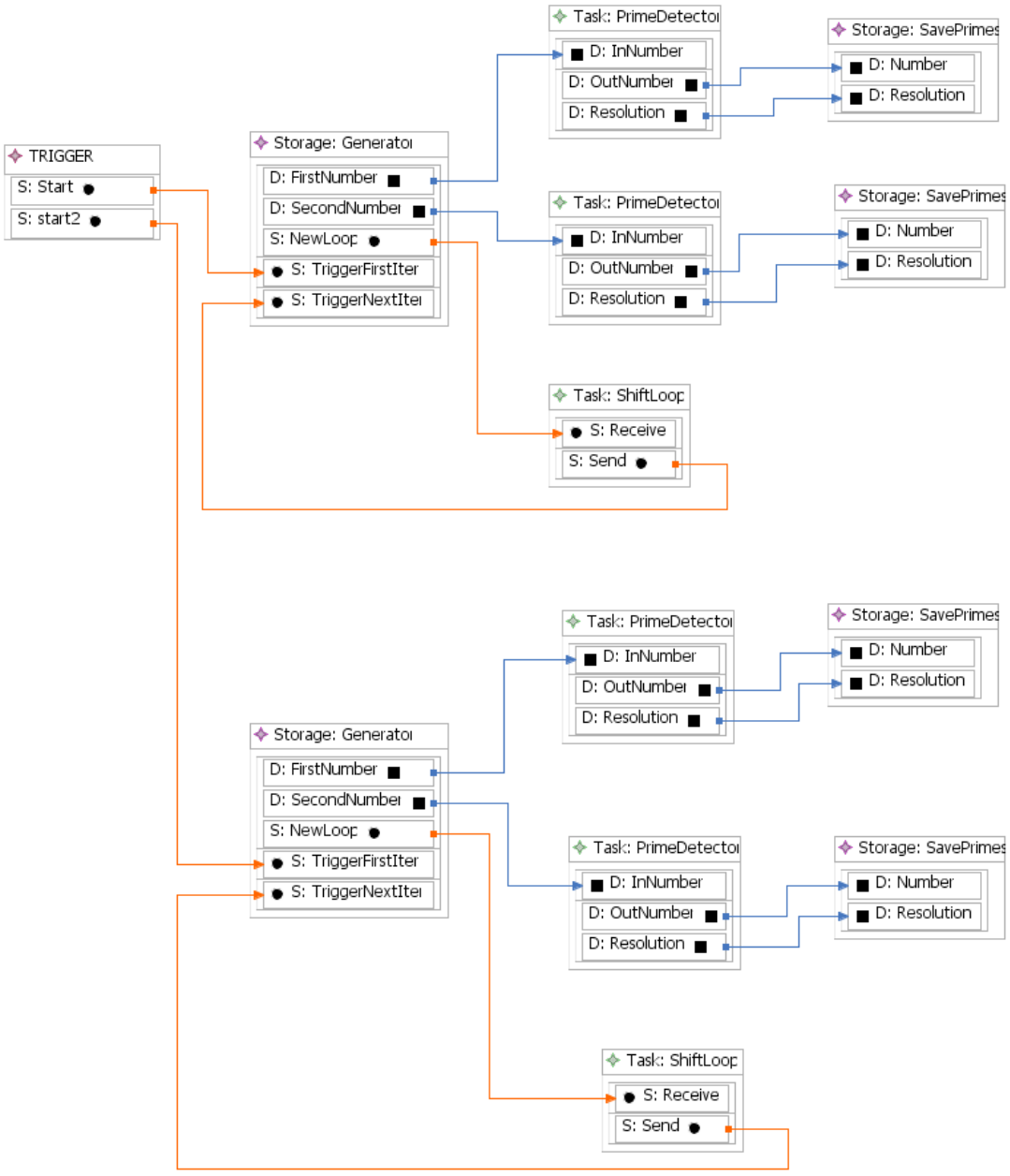



Figure C.2: Extended select of prime numbers

## C.3 Deadlock example

This example presents a capability of Activity Checker to recognise a deadlock state. See figure C.3. The name of the scheme file is *deadLockExample.dataflowscheme_diagram.*

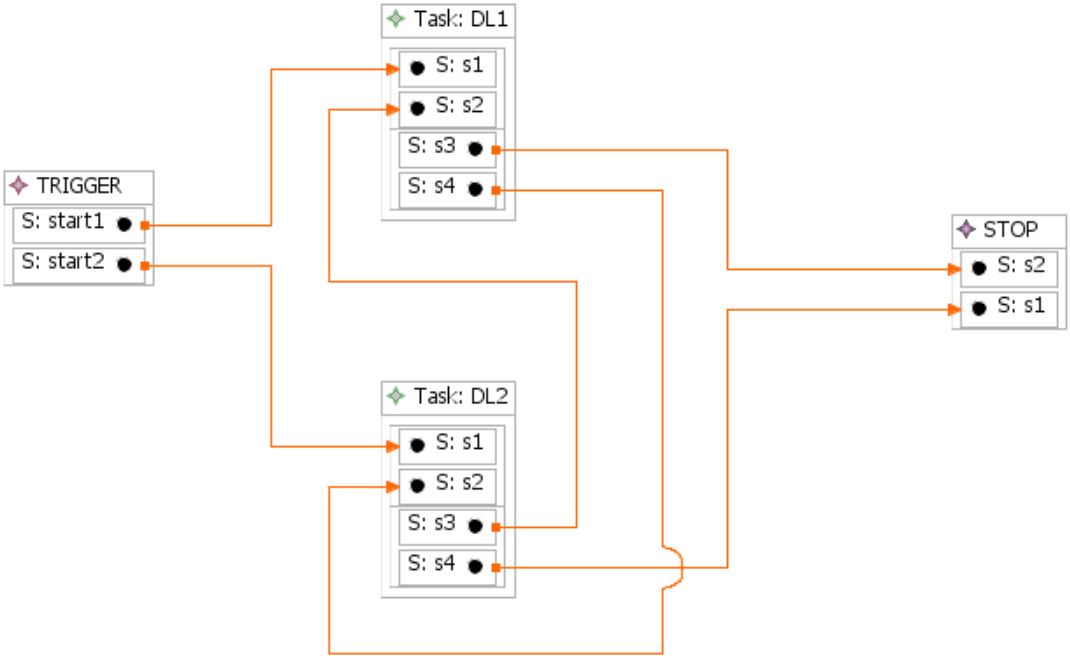



Figure C.3: Deadlock example